{EPITECH}

JAVA SEMINAR_

< DAY 08 - DESIGN PATTERNS />

JAVA SEMINAR

In 1995, the **Gang of Four** (GAMMA, HELM, JOHNSON & VLISSIDES) wrote a now very famous book: *Design Patterns: Elements of Reusable Object-Oriented Software* introduced 23 **design patterns**, which are ways to conceive and organize classes. Each design pattern is a reliable **language-independent** solution to a very common problem, a kind of good practice.

They are commonly used nowadays. More were added, and some variants appeared.
Let's study a few of them to extract some common principles, helpful to design applications.

To represent classes organization and content, UML diagrams will be used. You'll get a very brief introduction to these diagrams.

A brief introduction to design patterns and UML

The use of design patterns shows many advantages:

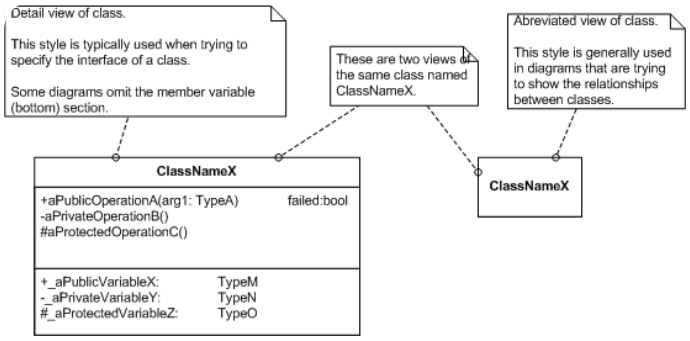
- ✓ **quality**: they are proven answers validated by experts ;
- ✓ **speed**: they are fast to implement and save time on conception brainstorming ;
- ✓ **reusability**: they can be reused in different applications without further development ;
- ✓ **ease**: they are fully documented and well-known ;
- ✓ **readability**: they are a common vocabulary for many people who use and master them.

Design patterns are subdivided into 3 categories:

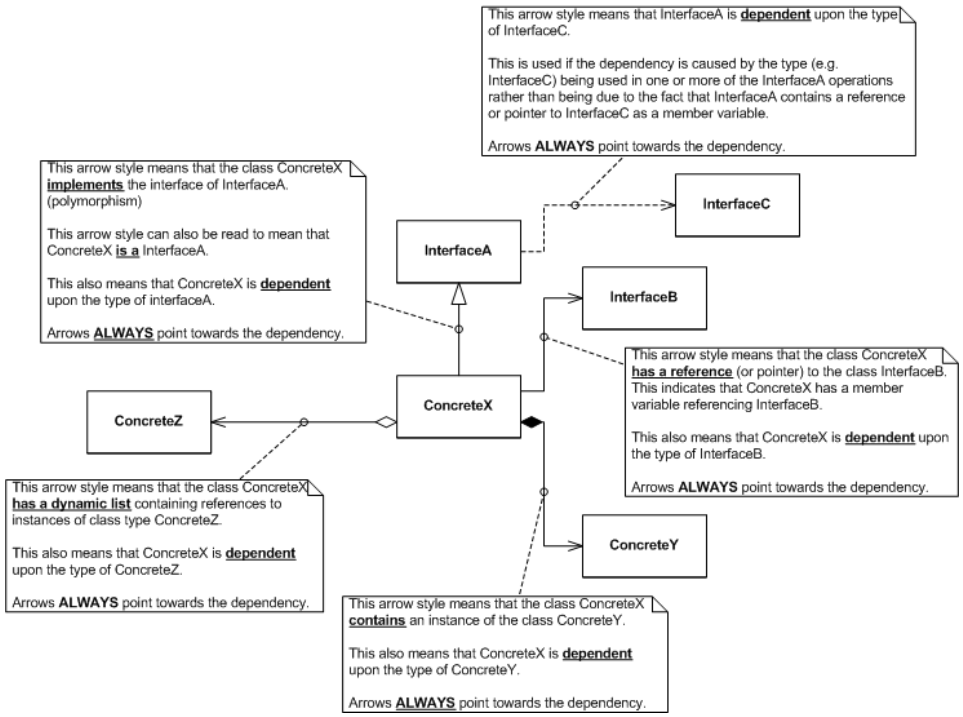
- ✓ **Creational**: to instantiate, initialize and configure classes and objects.
  Factory, *AbstractFactory*, *Builder* and *Prototype*.
- ✓ **Structural**: to organize and connect classes.
  Adapter, *Bridge*, *Composite*, *Decorator*, *Facade*, *Flyweight* and *Proxy*.
- ✓ **Behavioral**: to manage objects so that they can collaborate and interact.
  Interpreter, *Template Method*, *Chain of Responsibility*, *Command*, *Iterator*, *Mediator*, *Memento*, *Observer*, *State*, *Strategy* and *Visitor*.

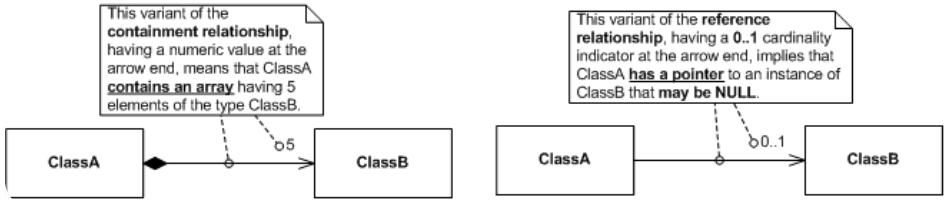
Design patterns are not always relevant. Some are lambasted by developers, probably rightly.

UML means **Unified Modeling Language**.
It is a uniformed and practical way to represent classes and their interactions.
A class is represented by a box like the one below:



Here are the different relations among them:





It might look a bit repellent at first glance, but it is not as tough as it seems.

More information

If design patterns are often handy, they are not always appropriate and are not the Alpha and Omega of OO conception. Rather than trying to apply (even cleverly) these patterns, understanding the object model in depth is way more relevant.

To do so, here is a list of concepts and Wikipedia references you should read:

✓ SOLID

- ***S**ingle responsibility*
  a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class).

- **O**pen/closed
  software entities... should be open for extension, but closed for modification.

- **L**iskov substitution
  objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

- **I**nterface segregation
  many client-specific interfaces are better than one general-purpose interface.

- **D**ependency inversion
  one should depend upon abstractions, not concretions.

SOLID is the keystone of conception, you should read some more about it:

✓ Design by contract
✓ Composition over inheritance

These are advanced principles, but you'll need to be familiar with them at some point.

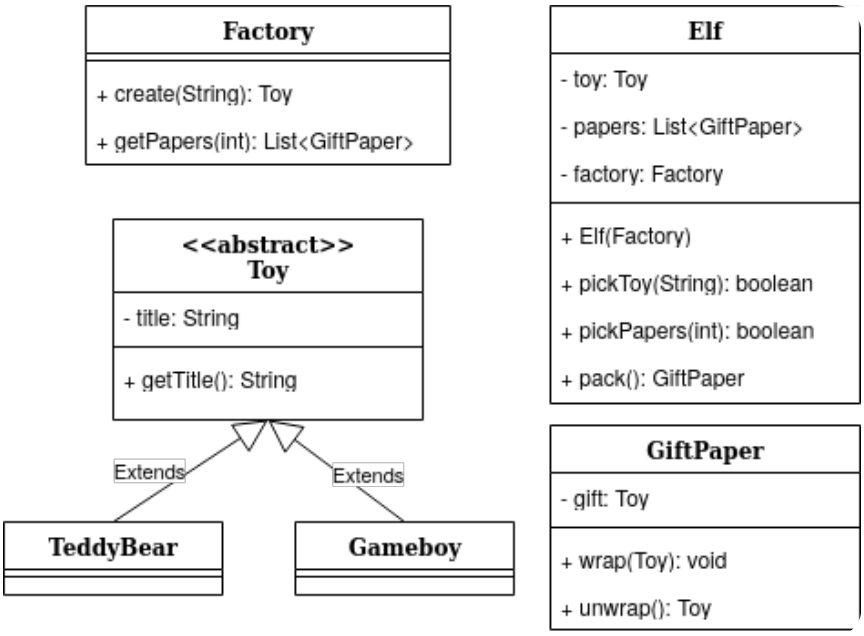
Exercise 01

Delivery: `./Factory/Toy.java`, `./Factory/TeddyBear.java`, `./Factory/Gameboy.java`, `./Factory/Factory.java`, `./Factory/GiftPaper.java`, `./Factory/NoSuchToyException.java`

All this exercise classes must be in the `Factory` package and must have a public visibility.

The **Factory** methods allow to encapsulate objects creations. This is useful when the creation process is complex, when it depends on configuration files or user entries for example.

Today Santa is asking you to manage his toy factory. Create the classes `Toy`, `TeddyBear`, `Gameboy`, `GiftPaper` and `Factory` with their attributes and methods as defined in the diagram below.



The `GiftPaper` has two methods: `wrap` that act as the attribute setter, and `unwrap` that both returns the attribut and set it to null.

The `Factory` class contains a `create` method:

- ✓ if the parameter is *teddy*, it returns a `TeddyBear` ;
- ✓ if it is *gameboy*, it returns a `Gameboy` ;
- ✓ else it must throw a `NoSuchToyException` with the following message: `No such toy: [toy name].`.

Finally, the `getPapers` method of your `Factory` creates a `List` containing `n GiftPaper`, where `n` is the number in parameter.

Exercise 02

Delivery: `./Factory/Toy.java`, `./Factory/TeddyBear.java`, `./Factory/Gameboy.java`, `./Factory/Factory.java`, `./Factory/GiftPaper.java`, `./Factory/NoSuchToyException.java`, `./Factory/Elf.java`

All this exercise classes must be in the `Factory` package and must have a public visibility.

Add to your classes the `Elf` class defined in the previous diagram.

✓ `pickToy`
The `Elf` will try to pick the corresponding Toy from the Factory.
He says `I didn't find any [toyName].` if this kind of toy doesn't exist.
The `Elf` cannot get something if he already has something in his hands.
This method returns `true` if, and only if, a new toy has been picked-up.
The method displays:

- `What a nice one! I would have liked to keep it...` if the `Elf` can get a toy.
- `Minute please?! I'm not that fast.` if he can't.

✓ `pickPapers`
Get `nb` pieces of `GiftPaper` from the Factory.
It always returns `true`.

✓ `pack`
Make the `Elf` pack the `Toy` he has in a `GiftPaper`, return it and say `And another kid will be happy!`
If the `Elf` has no Toy, instead it will say `I don't have any toy, but hey at least it's paper!`
If the `Elf` doesn't have `GiftPaper` anymore, it says `Wait... I can't pack it with my shirt.`
If there's no `GiftPaper`, the method will return `null`.

The Elf cannot directly create objects, it **must** use the factory to do so.

Exercise 03

Delivery: `./Composite/Sentence.java`, `./Composite/Word.java`, `./Composite/SentenceComposite.java`

All this exercise classes must be in the `Composite` package and must have a public visibility.

Create a `Sentence` **Interface** that only contains a `print` method.
This method takes no parameter and returns nothing.

Create a `Word` class that implements `Sentence` and override the `print` method to display the String that was passed to its constructor.

Create a composite class `SentenceComposite` that also implements `Sentence`.
It must contain a `List<Sentence>` named `childSentence` as attribute.
Override its `print` method to make it iterate on its children to call their own `print` methods.

Add the methods `add` and `remove` which both take a `Sentence` as parameter to add or remove a child.

Here is an example:

```java
public static void main(String[] args) {
    Word w1 = new Word("word1");
    Word w2 = new Word("word2");
    Word w3 = new Word("word3");
    Word w4 = new Word("word4");
    SentenceComposite sc1 = new SentenceComposite();
    SentenceComposite sc2 = new SentenceComposite();
    SentenceComposite sc3 = new SentenceComposite();
    sc1.add(w1);
    sc1.add(w2);
    sc1.add(w3);
    sc2.add(w4);
    sc3.add(sc2);
    sc3.add(sc1);
    sc3.print();
}
```

```
Terminal
$> java Example
word4
word1
word2
word3
```

Exercise 04

Delivery: `./Observer/Observable.java`, `./Observer/Order.java`, `./Observer/Observer.java`, `./Observer/Customer.java`

All this exercise classes must be in the `Observer` package and must have a public visibility.

Let's start an application for customers to see in real time the state of the orders they placed.
We'll use the **observer** design pattern.

This design pattern can be used when one object has several observers, when one observer has several objects to observe or when several objects are followed by several observers.

Create an `Observable` interface with two methods:

- ✓ `addObserver` : takes an `Observer` as parameter
- ✓ `notifyObservers` : returns a `boolean`, false if there is no observer.

Create a `Order` class which inherits from the `Observable` interface.

This class has four attributes: `position` (String), `destination` (String), `timeBeforeArrival` (int) and `observers` (List<Observer> that you will create later). Add a getter for the first three ones.

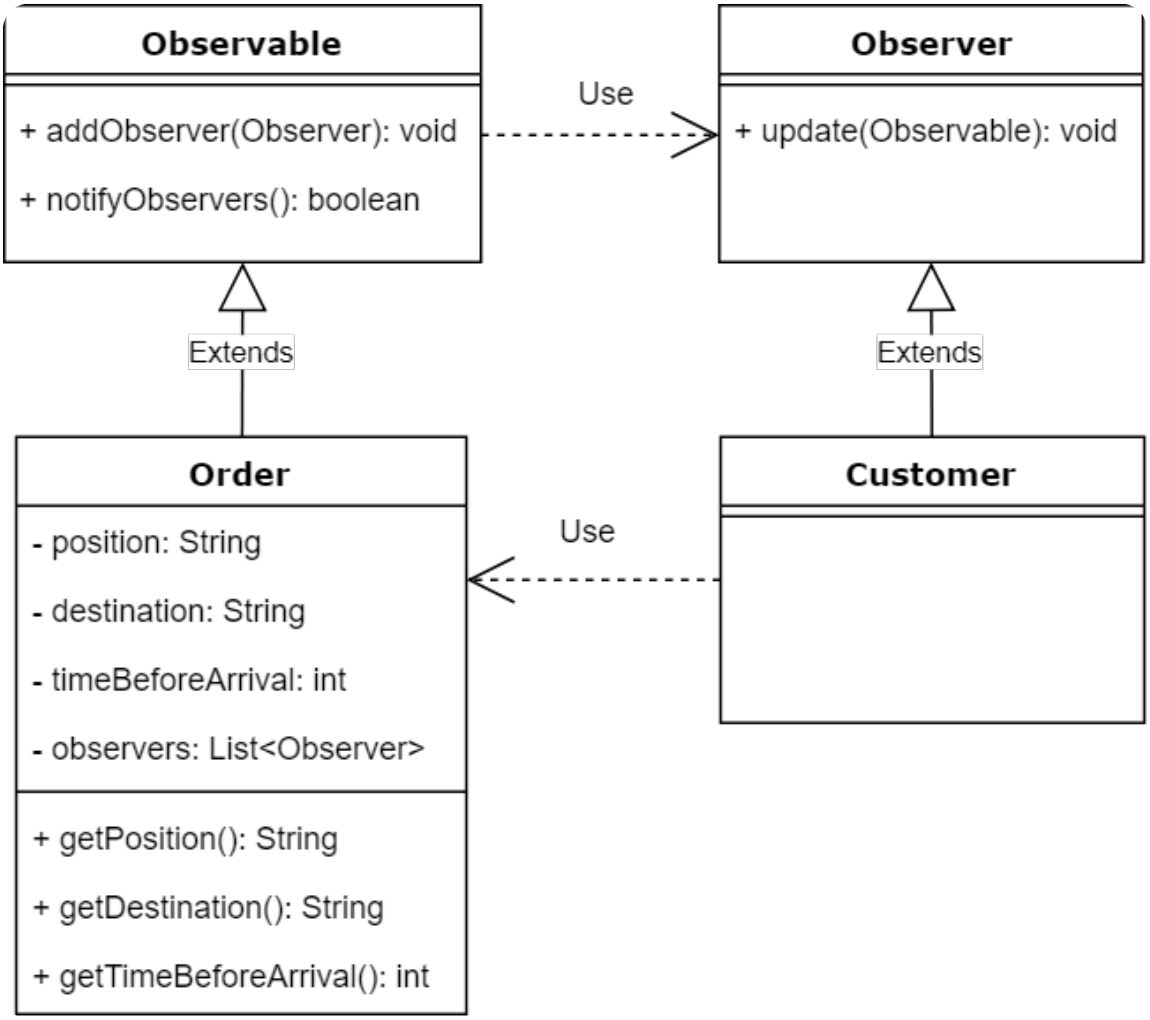
It also has a `setData` method which takes two strings and one int as parameters to set respectively the position, the destination and the time.

Then, create an `Observer` interface which only has one `update` method that takes an `Observable` as parameter.

Create now a `Customer` class which implements the `Observer` interface.
Its `update` method must display the delivery information the following way `Position ([position]), [time] minutes before arrival at [destination]`.

Use the **instanceof** keyword.

The `notifyObservers` method of your `Order` class should calls the `update` method of its `observers`.
This method must be called automatically after updating the delivery's data.



Let's see an example of how it should work:

```
public static void main(String[] args) {
    Order order = new Order();
    Customer customer = new Customer();

    order.addObserver(customer);
    order.setData("123.5326, 237.9277", "6W 40th Street, New York", 10);
    order.notifyObservers();
}
```

```
Terminal
$> java Example
Position (123.5326, 237.9277), 10 minutes before arrival at 6W 40th Street, New York.
Position (123.5326, 237.9277), 10 minutes before arrival at 6W 40th Street, New York.
```

Exercise 05

Delivery: ./Decorator/Warrior.java, ./Decorator/BasicWarrior.java, ./Decorator/KingWarrior.java, ./Decorator/StuffDecorator.java, ./Decorator/Shield.java, ./Decorator/FireSword.java

All this exercise classes must be in the `Decorator` package and must have a public visibility.

The **Decorator** design pattern is useful to enrich dynamically a basic class. Let's use it.

Create a **Warrior** abstract class with two attributes:

- ✓ an int, `hp`, for the health points ;
- ✓ an int, `dmg`, for the damage points it causes.

Add a getter only for each attribute: `getHp`, `getDmg`.

Now create two classes `BasicWarrior` and `KingWarrior` which inherit from `Warrior`, and set the following attributes in their constructors:

attribute	`BasicWarrior`	`KingWarrior`
`hp`	40	60
`dmg`	7	10

You now have to implement the decorator classes, whose goal is to add skills to your warriors.
Create the class `StuffDecorator` that inherits from `Warrior` and has a `Warrior` named `holder` as protected attribute.

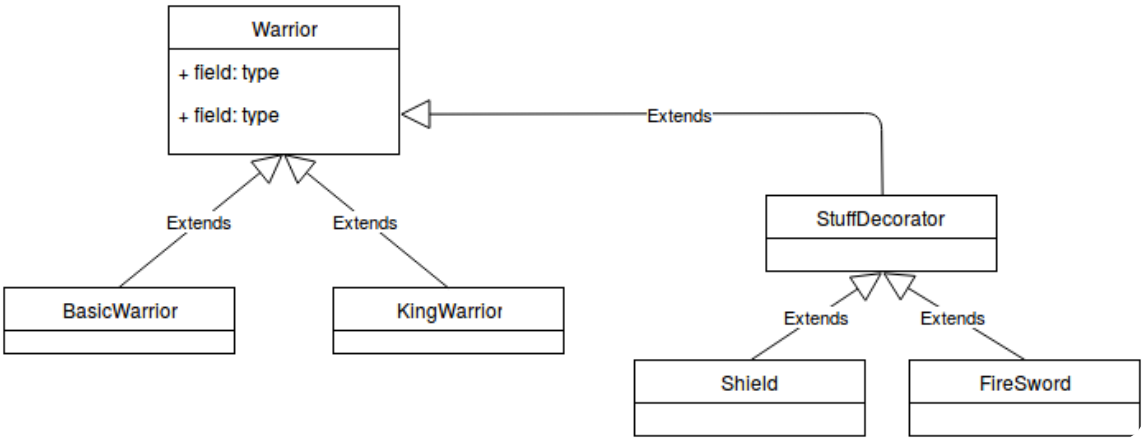
Make sure your `StuffDecorator` class overrides the getters from `Warrior` to return the value from `holder`.

Create a `Shield` class and a `FireSword` class which both inherit from the `StuffDecorator` class.
Their constructor takes a `Warrior` as parameter to initialize their attribute `holder`.

The `Shield` constructor displays `May this shield protect me against every enemy.`.
The `FireSword` displays `I can slice and burn like the wind and the flames.`.

Now override the getters to make:

- ✓ the `Shield` add 10 health points ;
- ✓ the `FireSword` 3 damage points.



Here is an example:

```java
public static void main(String[] args) {
    Warrior albert = new BasicWarrior();
    System.out.println("Albert has " + albert.getHp() + " health points.");
    albert = new Shield(albert);
    System.out.println("Albert has " + albert.getHp() + " health points.");

    Warrior georges = new KingWarrior();
    System.out.println("Georges has " + georges.getHp() + " health points and can hit " +
        georges.getDmg() + " damages.");
    georges = new FireSword(georges);
    georges = new Shield(georges);
    System.out.println("Georges has " + georges.getHp() + " health points.");
    System.out.println("Georges can hit " + georges.getDmg() + " damages.");
}
```

```
Terminal
$> java Example
Albert has 40 health points.
May this shield protect me against every enemy.
Albert has 50 health points.
Georges has 60 health points and can hit 10 damages.
I can slice and burn like the wind and the flames.
May this shield protect me against every enemy.
Georges has 70 health points.
Georges can hit 13 damages.
```

v 3.2.1